Tscope reference-manual

January 12, 2011

Abstract

This is the Tscope reference manual[1]. It gives an in-depth explanation of every function made available by Tscope.

Contents

[1] Copyright ©2003, 2004 Michael Stevens. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is is included in license.html.

1 Screen and double buffer operations

Defined in screen.h and screen.c

This part contains:

- A function to initialize the library and to set up a directX screen for the experiment to run in.
- Functions to change the setup of the screen.
- Functions to manipulate the double buffer.

1.1 int ts_init ();

Initializes the graphics mode and makes a double buffer if needed. The default settings are:

- Opens in a window.
- Resolution of 320 x 240 pixels.
- 16 bit color depth for fullscreen modes, same color depth as the desktop for window modes.
- 60Hz refresh rate for fullscreen modes (for window modes the refresh rate cant́ logically be different from the desktop refresh rate).
- No double buffer.

The default settings should be satisfactory while programming your experiment. The screen parameters can be changed with the functions ts_refreshrate, ts_colordepth, ts_scrsize, ts_scrmode en ts_doublebuff. These parameter functions have to be called before you call ts_init. With the next call of ts_init the requested values will be used.

Some of the most frequent changes to the default values are:

- full screen modes with a higher resolution during data collection.
- playing with refresh rates when doing priming experiments.
- using the double buffer for presenting complicated stimuli.

Most of the Tscope functions need the graphics screen to be opened (exceptions are the screen parameter functions themselves, randomizer functions, timer functions and sound stream functions). Therefore, any calls to these functions before you open the graphics screen with ts_init will trigger a graphics screen automatically.

Multiple instances of ts_init can be called throughout your program, for instance to change the refresh rate between two blocks.

If you successively change color depth and/or screen resolution, the current double buffer is destroyed and replaced by a new one. This means that the current buffer contents are removed and have to be reconstructed if desired. The destruction of the old buffer is necessary to avoid conflicts between old and new resolutions and/or color depths.

If the requested color depth is changed between two successive calls of ts_init, the foreground, background and mouse cursor colors are reset.

If opening a graphics screen fails, Tscope closes with an error message. This can happen for instance with some combinations of screen parameters that are not possible. The possibilities and limitations depend on the hardware used (load increases with increasing resolution, color depth and/or refresh rate). In the tests section, you will find a program called testscreen.c that can be used to determine which graphics modes are possible with your hardware.

While running in a window (e.g. during preliminary programming) the window is always kept smaller than the desktop. Furthermore, the window uses the color depth of the desktop for the sake of speed. Once going full screen, the actual requested settings will be used. If these settings are not realistic, the program quits with an error-message. The only exception to this behavior concerns the refresh rate. If the requested refresh rate is too high (we know that a refresh rate of 1000Hz would come handy), the program is not aborted but the refresh rate is automatically lowered. Tscope will inform you when it automatically lowers the requested refresh rate (see also ts_refreshrate).

The return value of ts_init is the actual refresh rate. This value and some more information about the operating system, desktop settings and the graphics mode are printed to the console.

1.2 void ts_fatal (const char *format, ...);

Closes the library safely, and prints the error-message provided in the format string *format to the console.

1.3 void ts_scrcfg (char *file)

Reads screen parameters from a configuration file.

A sample configuration file (tscope.cfg) is given in the examples section of the site.

The configuration file consists of sections which are marked by `[NAME OF SECTION]`. ts_scrcfg only reads the `[screen]` section. In this section there are five possible entries for setting the screen parameters. The entries are ts_debug=, ts_refreshrate=, ts_colordepth=, ts_scrsize= and ts_scrmode=.

Each entry controls what the parameter function with the same name would control.

All possible values for each entry are listed as comments in the sample configuration file. If a wrong value or more than one value is entered for a given entry tscope closes with an error message.

If one of the entries is not in the file (or commented out with #) the corresponding parameter will not be altered by ts_scrcfg.

If an entry appears more than once in the file, only the first appearance has effect.

The screen is not automatically opened or re-opened by this function.

1.4 int ts_debug(int mode);

Controls the amount of debugging info that is printed on the console while Tscope is running. The default setting is DEBUG1. There are five possible settings:

- DEBUG0 no debugging info
- DEBUG1 report automatic changes to screen parameters
- DEBUG2 report when Tscope subsystems are loaded
- DEBUG3 print some info about the system we're running on
- DEBUG4 report when data files and response keys are loaded

If other values are passed to this function, Tscope closes with an error message.

Returns previous setting.

1.5 int ts_refreshrate (int rate);

Changes the refresh rate to be requested with the next call of ts_init. The default value is 60 Hertz.

There are limitations in the use of refresh rates. The possibilities depend on a combination of the hardware, the color depth and the resolution you want to use. If the refresh rate you asked for is beyond the possibilities of your system, the closest possible rate is used. When working in a window (not fullscreen) the refresh rate of the desktop is used. The eventual used refresh rate is returned with the next call of ts_init.

Frequently used refresh rates are 60, 70, 72, 75, 85 and 100. All values between 50 and 150 will be accepted by ts_refreshrate, but remind that this doesn't necessarily mean that your hardware supports these values. When feeding this function with values below 50 or above 150, Tscope closes with an error message.

Returns the previous refresh rate.

1.6 int ts_colordepth (int depth);

Changes the color depth to be requested with the next call of ts_init. The default color depth is 16 bit. When working in a window (not fullscreen) the color depth adopts to the color depth of the desktop for the sake of speed.

Possible values are 8, 15, 16, 24 or 32 bits. If other values are passed to this function, Tscope closes with an error message.

Returns previous color depth.

1.7 int ts_scrsize (int size);

Changes the screen resolution to be requested with the next call of ts_init. The default resolution is 320 x 240 pixels. When working in a window, the dimensions of the window are always kept smaller than the dimensions of the desktop.

Possible dimensions are given below, together with their industry-standard name - which can also be used in calls to ts_scrsize - and corresponding resolution.

- SIZE0 (QVGA, 320 x 240 pixels).
- SIZE1 (VGA, 640 x 480 pixels).
- SIZE2 (SVGA, 800 x 600 pixels).
- SIZE3 (XGA, 1024 x 768 pixels).
- SIZE4 (XGAplus, 1152 x 864 pixels).
- SIZE5 (SXGA, 1280 x 1024 pixels).
- SIZE6 (SXGAplus, 1400 x 1050 pixels).
- SIZE7 (UXGA, 1600 x 1200 pixels).

Sizes 3 to 7 also have a widescreen version:

- WSIZE3 (WXGA, 1280 x 768 pixels).
- WSIZE4 (WXGAplus, 1440 x 900 pixels).
- WSIZE5 (WSXGA, 1600 x 1024 pixels).
- WSIZE6 (WSXGAplus, 1680 x 1050 pixels).
- WSIZE7 (WUXGA, 1920 x 1200 pixels).

For those interested in the weird industry-standard names: VGA stands for Video Graphics Array, QVGA for Quarter VGA, SVGA for Super VGA, XGA for eXtended Graphics array, SXGA for Super XGA, UGXA for Ultra XGA. The W means: Widescreen.

If other values are passed to this function, Tscope closes with an error message.

If a fullscreen resolution is too big for your monitor or graphics adaptor, Tscope also closes with an error message.

Returns the previous resolution.

1.8 int ts_scrxy (int x, int y);

As an alternative to ts_scrsize, this function can be used to specify the width and height of the screen separately (in number of pixels on the x and y axis). Only use this if none of the resolutions that can be set using ts_scrsize match your requirements.

Returns the previous resolution.

1.9 int ts_scrmode (int mode);

Sets the screen mode. Default is WINDOW. WINDOW opens a window, FULLSCREEN opens fullscreen without hardware acceleration, FULLSCREEN_ACCEL opens fullscreen with hardware acceleration. If other values are passed to this function, Tscope closes with an error message.

Returns the previous screen mode.

1.10 int ts_doublebuff (int mode);

Sets the double buffer setting to be requested with the next call of ts_init. Default is OFF. ON means working with double buffer, OFF without. If other values are passed to this function, Tscope closes with an error message.

The double buffer can be used if your stimuli are too complex to draw within one refresh cycle of the screen. In that case, you take the time necessary to draw your stimuli on the buffer, and after it's finished you can blit the buffer onto the screen within a few milliseconds.

The double buffer will be automatically destroyed and replaced by a new one if the color depth or resolution of the screen are changed.

Returns the previous double buffer setting.

1.11 void ts_clrscr();

Clears the screen. If the screen is not running yet, Tscope closes with an error message.

1.12 void ts_scrdump ();

Grabbing function that makes a dump of the screen to a file in the present working directory. The first screendump made by a program will be called dump1.bmp, the second dump2.bmp, etc. If the screen is not running yet, Tscope closes with an error message.

1.13 void ts_tobuff ();

When calling drawing or text output operations, Tscope needs to know the destination (screen or double buffer). The default destination is the screen. This function makes the double buffer the destination. If the screen is not running yet, or no double buffer is present, Tscope closes with an error message.

1.14 void ts_toscr ();

Sets the screen as destination bitmap for drawing. This function is necessary when working with a double buffer or memory bitmaps. If the screen is not running yet, Tscope closes with an error message.

1.15 void ts_clrbuff ();

Clears the double buffer. If the screen is not running yet, or no double buffer is present, Tscope closes with an error message.

1.16 void ts_blitbuff ();

Blits the double buffer to the screen. If the screen is not running yet, or no double buffer is present, Tscope closes with an error message.

2 Graphics parameters

Defined in graphics.h and graphics.c

This part contains functions to change the graphics parameters of the text output and drawing functions. By default, text appears in a terminal font, and circles, ellipses etc. are not filled. All drawing is done in white against a black background. The text and drawing functions do not have arguments to alter the graphics parameters. This is done with the parameter functions below. They all change one parameter, e.g. the foreground color. Once a parameter value is changed, the given value is used for all further drawing until it is changed again.

The parameter functions all have the previous parameter setting as return value. This can be useful when you want to change some parameters in one part of a program, do some drawing and set the previous parameters again, without causing undesired parameter changes to other parts of the program.

2.1 int ts_bgcolor (int color);

Sets the background color. Default is black. Once a color is set, it remains active until another color is set. When this function is called, both the screen and double buffer are cleared to match the new background color.

Tscope represents colors as integer values. You can use predefined color values (see appendix) or you can compute a color value based on RGB triplets with ts_makecolor.

Returns the previous background color value.

Possible colors can be found in the appendix of the reference manual, or can be made by yourself through ts_makecolor.

2.2 int ts_fgcolor (int color);

Changes the drawing color of all subsequent drawing operations. The default color is white. Once a color is set, it remains active until another color is set.

Tscope represents colors as integer values. You can use predefined color values (see appendix) or you can compute a color value based on RGB triplets with ts_makecolor.

Returns the previous drawing color value.

2.3 int ts_textmode (int color);

Sets the background color of text. Default is transparent (predefined as TRANS). Once a color is set, it remains active until another color is set.

Tscope represents colors as integer values. You can use predefined color values (see appendix) or you can compute a color value based on RGB triplets with ts_makecolor.

Returns the previous text background color value.

2.4 int ts_makecolor (int r, int g, int b);

Computes a color value based on an RGB triplet. r, g and b are from the range [0,255].

It is possible to call this function as a parameter to the functions ts_bgcolor, ts_fgcolor or ts_textmode (e.g. ts_fgcolor (ts_makecolor (255,0,0)); changes the drawing color to red).

2.5 int ts_makehsvcolor(float h, float s, float v);

Computes a color value based on an HSV (hue, saturation, value) triplet. Hue is between 0 and 360, saturation and value between 0 and 1.

This function can be used as an alternative to ts_makecolor.

2.6 int ts_fill (int mode);

This function can be used to set the fill-mode for future drawing operations. When set to ON, circles and rectangles, etc. are filled. Only contours are drawn when set to OFF. Default is OFF.

Returns the previous fill-mode.

2.7 int ts_font (int type, int size, int style);

Sets the font used for future text output operations. Three fonts are available: ARIAL, COURIER and TIMES. They all come in 16 sizes: 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48 and 72 pixels. Four styles are possible: regular (REG), and bold (BOLD), italics (IT), or bold italics (BOLDIT).

Default a terminal-like font is used. The other fonts are provided in separate data files and are loaded the first time they are used. This takes some time (not exactly much, but enough to mess up the timing of a response), so make sure all used fonts are loaded before the first trial starts.

The font data files are provided in `/usr/local/fonts`. Tscope will search for the font files in the program's working directory, the `fonts` subdirectory of this directory and in `/usr/local/fonts`.

Returns the number of the previous font.

2.8 int ts_setfont (int fontno);

Sets the font based on the return value of a previous call to ts_font. Can be useful if one part of a program changes the font and wants to switch to the previous font again, without explicitly knowing what the previous font was.

ts_setfont(0) switches back to the default terminal font.

Returns the number of the previous font.

2.9 int ts_loadfont (char *fontfile);

Sets a font from a custom font file. It will search for the font file in the program's working directory, the `fonts` subdirectory of this directory and in `/usr/local/fonts`.

The Tscope FAQ explains how to make a custom font file.

Returns the number of the previous font.

3 Coordinate system

Defined in coordinates.h and coordinates.c

There are two coordinate systems available in Tscope. You can use the standard computer coordinates, with the origin (0,0) in the upper left corner of the screen, with increasing coordinate values to the left of and below the origin. You can also use an Cartesian system, with the origin (0,0) in the center of the screen and increasing coordinate values to the left and above the origin, and decreasing (negative) values to the right and below the origin. Default is the Cartesian system.

Coordinates are given in pixels. Helper functions are available that transform percentages of the screen size into number of pixels.

The Cartesian system makes simple stimulus setups in the center of the screen simpler, but not really suitable for more complex operations like blitting, or for experienced graphics programmers. Switching between the systems can be done with a parameter function.

3.1 XMAX

Macro that contains the horizontal size of a screen quadrant in pixels. Its value is not defined before the graphics screen is opened.

3.2 YMAX

Macro that contains the vertical size of a screen quadrant in pixels. Its value is not defined before the graphics screen is opened.

3.3 SXMAX

Macro that contains the horizontal size of the screen in pixels. Its value is not defined before the graphics screen is opened.

3.4 SYMAX

Macro that contains the vertical size of the screen in pixels. Its value is not defined before the graphics screen is opened.

3.5 int ts_coordinates (int system);

Switches between the Cartesian and standard coordinate system. Predefined values are CARTESIAN and STANDARD. Default is CARTESIAN.

Returns the value of the previous setting.

3.6 int ax (float x);

Transforms percentages of the horizontal screen or screen quadrant size to absolute values (pixels). Its behavior depends on the coordinate system used.

With the Cartesian system x=0 refers to the center of the screen, x=1 refers the to right-hand side border and x=-1 refers to the left-hand side border.

With the standard system x=0 refers to the right-hand side border, x=1 refers to the left-hand side border.

It is possible to call this function as a parameter to a draw-function (e.g.: when using the Cartesian coordinate system, ts_putpixel (ax(.5), 0); writes a pixel eighty pixels to the right of the center in the case of a 320x240 sized screen, or 160 pixels to the right in the case of a 640x480 sized screen).

3.7 int ay (float y);

Like ax, but for horizontal coordinates.

3.8 int sx (float x);

Converts horizontal Cartesian coordinates to standard coordinates.

3.9 int sy (float x);

Converts vertical Cartesian coordinates to standard coordinates.

3.10 int cx (float x);

Converts horizontal standard coordinates to Cartesian coordinates.

3.11 int cy (float y);

Converts vertical standard coordinates to Cartesian coordinates.

3.12 void ts_agrid ();

Useful function when designing the graphical interface of your experiment. Draws a Cartesian/standard grid to the screen using absolute coordinates (pixels). This function is especially handy when you want to obtain absolute pixel sizes of all your stimuli (dependent of screen size).

3.13 void ts_rgrid ();

Useful function when designing the graphical interface of your experiment. Draws a Cartesian/standard grid to the screen using relative coordinates (percentages of screen/quadrant size). This function is especially handy when you want to obtain relative sizes of all your stimuli (independent of screen size).

4 Formatted text input/output

Defined in textio.h and textio.c

The text input-output functions all take the smallest possible number of parameters to put something on the screen: two coordinates, and a format string.

The vertical coordinates are interpreted differently by the two coordinate systems. With Cartesian coordinates, the text is vertically centered around the y coordinate. With standard coordinates, the y coordinate corresponds to the top of the text (like in other libraries).

The format string is the same as the standard C printf and scanf function families. It does not recognize the `\t` and `\n` escape characters, however. These are rendered as a a caret (^).

Drawing parameters are set with the graphics parameter functions.

4.1 int ts_printf(int x, int y, const char *txt, ...)

Writes text in *txt to position x, y of the current destination bitmap. The color of the text is defined with ts_fgcolor. The string to be printed can be specified using a printf() like format string. The maximum length of the string is set to 512 characters.

Returns the length of the output string in pixels.

4.2 int ts_printf_centre(int x, int y, const char *txt, ...)

Like ts_printf, but centered around x, y.

Returns the length of the output string in pixels.

4.3 int ts_printf_right(int x, int y, const char *txt, ...)

Like ts_printf, but aligned to the right of x, y.

Returns the length of the output string in pixels.

4.4 int ts_printf_justify(int x1, int x2, int y, const char *txt, ...)

Like ts_printf, but the text is justified between x1 and x2.

Returns the length of the output string in pixels.

4.5 int ts_textheight ()

Returns the height of the font in pixels.

4.6 int ts_textlength (char *txt)

Returns the length of string *txt in pixels.

4.7 int ts_scanf (int x, int y, char *format, ...);

Reads input from the keyboard with formatted input like scanf(). The input appears on the screen starting at position (x, y).

By default, the standard US keyboard layout (qwerty) is used. If you want to use another keyboard layout, you will have to reconfigure Allegro on a per-computer basis. The Tscope FAQ explains you how.

Returns the number of variables that have been read successfully, or EOF (-1) if no input has been read at all. If reading fails, the content of the supplied variables is not altered. Therefore it is advisable to initialize any variables supplied to ts_scanf before calling the function, and/or to check the return value of ts_scanf after the call.

5 Drawing primitives

Defined in draw.h and draw.c

Like the text input-output functions, the drawing functions all take the smallest possible number of parameters to put something on the screen: some coordinates.

Drawing parameters are set with the graphics parameter functions.

5.1 void ts_getpixel (int x, int y)

Reads the color of the pixel at position (x,y). The source bitmap is set using ts_tobuff, ts_toscr or ts_tobmp.

5.2 void ts_putpixel (int x, int y)

Writes a pixel to the specified position in the bitmap. Color set with ts_fgcolor and the destination bitmap is set using ts_tobuff, ts_toscr or ts_tobmp.

5.3 void ts_circle (int x, int y, int radius)

Draws a circle around point (x,y) using radius as radius. Use ts_fill for switching fill mode.

5.4 void ts_line (int x1, int y1, int x2, int y2)

Draws a line between (x1,y1) and (x2,y2).

5.5 void ts_hline (int x1, int x2, int y)

Draws a horizontal line between x1 and x2 at vertical position y.

5.6 void ts_vline (int x, int y1, int y2)

Draws a vertical line between y1 and y2 at horizontal position x.

5.7 void ts_rect (int x1, int y1, int x2, int y2)

Draws an outline rectangle with the two points as its opposite corners. Use ts_fill for switching fill mode.

5.8 void ts_triangle (int x1, int y1, int x2, int y2, int x3, int y3)

Draws an outline triangle with points (x1,y1), (x2,y2) and (x3,y2) as vertexes. Use ts_fill for switching fill mode.

5.9 void ts_polygon (int points, int *xy)

Draws a polygon with an arbitrary number of vertexes. Pass the number of vertexes and an array containing a series of x, y points (should be an array with a total of vertexes*2 values). Polygons are never filled. Only the contours are drawn.

5.10 void ts_arc (int x, int y, int ang1, int ang2, int r);

Draws an arc with center (x,y) and radius r starting at angle ang1 until it reaches angle ang1. Angles are in degrees counter-clockwise (0-360): zero is to the right of the center point, larger values rotate anticlockwise from there.

5.11 void ts_ellipse (int x, int y, int rx, int ry);

Draws an ellipse with center (x,y) and x-radius rx and y-radius ry. This function is only suitable for horizontally or vertically oriented ellipses. No other rotations are supported. Use ts_fill for switching fill mode.

5.12 void ts_floodfill (int x, int y);

Floodfills an enclosed area with the current drawing color starting at point (x,y).

6 Timing and registering responses

Defined in timer.h and timer.c

This part contains:

- Functions to define response buttons.
- Functions to register stimulus and response times (in clock tics) and to estimate the timing error (i.e. the time elapsed between successive checks of the input status).
- Functions to convert clock tics to microseconds, milliseconds and seconds, and the other way around.
- A function to set the program priority.

While most other function groups in Tscope only use Allegro and standard C functions, this function group pulls together three different libraries. In case you want to program a custom timing function, you will need to know the following:

- *windows.h* is used to read out the system timer and sets program priority.
- *ioperm.h* is used to read out the status of the game port, parallel port, and vsync registry with millisecond accuracy.
- *allegro.h* is used to read out joystick, keyboard and mouse status. Because of the type of buttons used in these devices, they can not be used for millisecond accurate response registration.

6.1 void ts_timercfg (char *file)

Reads timing parameters from a configuration file.

A sample configuration file (tscope.cfg) is given in the examples section of the site.

The configuration file consists of sections which are marked by `[NAME OF SECTION]`. ts_timercfg only reads the `[timing]` section. In this section there are five possible entries for setting the timing parameters. The entries are ts_priority=, ts_waitmode=, ts_vsyncmode=, ts_vsynclimit= and ts_defkey=.

Each entry controls what the parameter function with the same name would control.

All possible values for each entry are listed as comments in the sample configuration file. If a wrong value or more than one value is entered for a given entry tscope closes with an error message.

The ts_defkey= entry is an exception to this: here, more than one parameter (response key) can be set. All of the response key should be listed on one line, separated by spaces. The first key listed gets number 1, the second gets number 2, etc.

If one of the entries is not in the file (or commented out with #) the corresponding parameter will not be altered by ts_timercfg.

If an entry appears more than once in the file, only the first appearance has effect.

When calling this function, all existing response key definitions are removed and the timing subsystem is automatically restarted.

Reads timing parameters from a configuration file.

6.2 int ts_priority (int prio)

Sets the program priority. Six values are defined: REALTIME, HIGH, ABOVE_NORMAL, NORMAL, BELOW_NORMAL and IDLE. By default, Tscope programs run with NORMAL priority.

Returns value of previous priority setting.

Be careful with higher priority modes. To avoid problems, set higher priority modes only when needed (during a block of trials) and switch back to normal priority when timing accuracy is not an issue.

Only start increasing the priority when you are sure your program works without any problems. HIGH priority is the best compromise between timing accuracy and safety. A program that runs with HIGH priority will not be put aside by normal programs, but it will still be possible to stop the program using the task manager.

Only use REALTIME priority when the other priority modes are not sufficient. With realtime priority, an endless loop in your program will hang the system - there is no way to stop a program running with REALTIME priority!.

Also keep in mind that reading input from keyboard, mouse and joystick is impossible when running with REALTIME priority on Windows XP. Playing .wav files will not be possible either when running with REALTIME priority.

The BELOW_NORMAL and IDLE priority modes are not really useful for running experiments. For running simulations they are: in IDLE mode, a program will only use system resources if there are no other processes running (i.e. when you're not using your computer for anything else).

6.3 __int64 ts_time();

Returns time in clock tics since computer start-up. This function can be useful to check the speed of a particular piece of code. It can also be used for custom timer functions (double task, ...). Use this function only if needed. For regular response registration it's easier to use ts_resp.

6.4 void ts_wait(__int64 time);

Wait function. This function uses a regular busy-loop, but it keeps monitoring input from response devices. This is useful to avoid that responses registered during the inter trial interval are buffered, causing the next response registration to "flip through".

6.5 int ts_waitmode(int mode);

Alters the waiting behavior of Tscope's timing functions. Normally these functions run in tight busyloops that never return control to the operating system. This way (almost) all cpu time will be used by your Tscope program. During experimental runs this is the best choice (best timing precision), but during development and testing of a program it can be annoying (as other programs will run very slow).

ts_waitmode controls under which conditions Tscope will explicitly give up cpu time and allow other programs to run. There are four possible parameters:

- BUSYLOOP: never give up cpu time (default - best timing precision).
- SLEEP_WAIT: give up cpu time (sleep) while waiting for a time interval to elapse. Cpu usage drops but timing accuracey of ts_wait will be less.
- SLEEP_RESP: give up cpu time while waiting for an interval or a response. Cpu usage drops even more but timing accuracy of ts_resp and ts_release will also be less.
- SLEEP_VSYNC: give up cpu time while waiting for an interval, a response or a vertical screen sync. Cpu usage drops a tiny bit further but ts_vsync can completely stop working. Don't use this parameter: it's not here for actual use, only for the sake of completeness.

Returns the value of the previous waiting mode.

6.6 int ts_defkey (int key)

Defines the response keys. They should be entered one by one, by repeating the call to this function. The first key defined gets number 1, the second defined 2, etc.

Returns the response number assigned to the key.

All key codes can be found in the appendix. To remove the whole key definition use ts_clrkeys (see infra). To temporarily hide one key from the list of defined keys, use ts_hidekey (see infra).

At present Tscope can read the parallel port and game port with millisecond accuracy. Reading the keyboard, mouse and joystick is also possible but with limited precision. Don't use these under realtime priority. Use the parallel and game port during the actual experiment. Use the mouse, keyboard or joystick while developing. Try to avoid input from multiple devices. This can cause delays.

All necessary permission settings and driver requests are automatically handled at start-up and shutdown of your program. When loading a driver fails, the program will stop with an error message.

6.7 int ts_hidekey(int key);

Temporarily deactivates one response key, which is useful in situations where multiple simultaneous keypresses are registered (see timing examples).

The argument to this function is the response value that has to be deactivated, not the key code.

Reactivating all defined keys is possible by calling ts_hidekey(0).

Returns the number of active keys left. If no active keys are left, Tscope closes with an error message (to avoid endless loops).

6.8 void ts_clrkeys();

Removes the whole response key definition.

6.9 int ts_resp(__int64 *time, __int64 *error, __int64 maxtime);

Waits for a response. Writes the number of tics between computer start-up and the response to *time, writes the estimated timing error to *error. Returns the number of the response key pressed.

The value of maxtime defines the maximum time to wait for a response (in tics). Setting maxtime to 0 means waiting until response, however long. If no response is registered at the end of the interval 0 is returned and the time and timing error at the end of the interval are written to *time and *error.

When no keys are defined, this function closes Tscope with an error message (to avoid endless loops).

6.10 int ts_release(__int64 *time, __int64 *error, __int64 maxtime);

Waits until all response buttons are released. Writes the number of tics between computer start-up and the release to *time, writes the estimated timing error to *error. The function returns the number of the response key pressed (should always return 0, meaning that all buttons are released).

The value of maxtime defines the maximum time to wait for a release (in tics). Setting maxtime to 0 means waiting until release, however long. When no keys are defined, this function closes Tscope with an error message (to avoid endless loops).

6.11 int ts_respstatus();

Returns the status of the response keys. Use this function when there is need for a custom timer function. Take care though: to maximize speed the function doesn't check if Tscope is actually running, nor if any response keys are defined. Take a look at the code of ts_resp to find out what security measures you need and where to place them.

6.12 void ts_flushresp();

Waits for release of all response buttons.

6.13 void ts_vsync(__int64 *time, __int64 *error);

Waits for a vertical retrace. Writes the number of tics between computer start-up and the moment of the sync-signal to *time, writes the estimated timing error to *error.

Under ideal circumstances, the function reads the status of vertical retrace signal directly from the computer's registry. In situations where that's not possible, a vertical retrace signal is simulated.

Tscope will simulate the retrace signal at the actual refresh rate in the following cases:

- In windowed mode.
- If the refresh rate is too high. Default the maximum value is 70Hz, but it can be altered by the function ts_vsynclimit.
- If the vsync register cannot be opened. (i.e. if ioperm is not properly installed).

If no window is opened yet, Tscope will simulate the retrace at the requested refresh rate.

With every change of the graphics mode, the vsync mode will be updated.

Note that the reliability of the readout or simulation of the vertical retrace is highly influenced by the priority of the program. Reliability increases with the priority of the program (but be aware of the possible problems with realtime priority).

6.14 void ts_vsyncs (__int64 *time, __int64 *error, int nsync);

Adapted ts_vsync. Waits until the beginning of the nsync-th vsync. Can be useful in priming-experiments (e.g.: prime remains on screen for 3 refresh cycles).

Writes the number of tics between computer start-up and the moment of then nsync-th sync-signal to *time, writes the estimated timing error to *error.

6.15 int ts_vsyncresp(__int64 *time, __int64 *error, int maxsync);

Like ts_resp, but waits for a given number of vsyncs instead of milliseconds. Use this function when you want the stimulus to be removed in a proper way (synchronized with the screen) during the response interval.

6.16 int ts_vsyncstatus();

Returns the status of the vsync signal. Returns 1 at the beginning of a screen retrace, 0 otherwise. Use this function when there is need for a custom timer function. Take care though: to maximize speed the function doesn't check if Tscope is actually running, nor if the vsync system is started. Take a look at the code of ts_vsync to find out what security measures you need and where to place them.

6.17 int ts_vsyncmode(int mode);

This function is written for the sake of completeness, not for actual use. Use it at your own risk.

By default, Tscope chooses between reading out the vsync signal from the registry, simulating a vsync signal at the requested refresh rate, or simulating a vsync signal at the actual refresh rate. With this function, you can force Tscope to choose one of the alternatives.

Parameter values are:

- REALSYNC will force Tscope to read out the vsync registry.

- SIMSYNC_ACTUAL will force Tscope to simulate vsync at the actual refresh rate.
- SIMSYNC_REQ will force Tscope to simulate vsync at the requested refresh rate.
- WHATEVER will leave the choice to Tscope. Default.

Forcing Tscope to read out the vsync signal from the registry can result in endless loops, if the signal cannot be caught. Forcing Tscope to simulate the vsync signal at the requested refresh rate will result in bad estimates of how long your stimulus was on screen, when the requested and actual refresh rate differ.

This function returns the previous vsync mode.

6.18 int ts_vsynclimit(int limit);

Changes the refresh frequency limit where Tscope will start using vsync simulation. Default this is set to 70Hz. Above this frequency, the vsync signal cannot be read out fast enough on most systems, resulting in signal misses.

On some systems, this frequency limit might be higher. Only change this value if you are confident that the new value will work on your system. Expect endless loops if you set the value too high.

All values between 50 and 150 will be accepted by ts_vsynclimit (this doesn't mean that your hardware supports these values).

This function returns the previous vsync limit.

Use this function at your own risk.

6.19 int ts_setserialport(int port);

Parameter function to specify on which serial port a Cedrus RB-x30 Response Pad is configured. The port number can be found under Control Panel - System - Hardware - Device Manager - Ports, or by running tscope/tests/testsport.c

This function needs to be called before defining response keys on the Cedrus response pad (S1-S7).

This function returns the previous serial port number.

6.20 void ts_settrigger (int port, __int64 time);

Parameter function to send trigger signals to the parallel port of a computer (for syncronizing your Tscope-program with programs running on another computer, e.g. an ERP-machine or some other imaging device).

Port contains the number of the parallel port you want to write to (1, 2 or 3), time contains the number of tics each trigger has to stay on (depends on the timing resolution of the computer you're sending triggers to).

6.21 void ts_trigger (char val);

Sends a trigger value to the parallel port specified by ts_settrigger. The signal is reset to 0 after the time specified with ts_settrigger.

Trigger parameters have to be set before you call this function. Tscope will close with an error message if you don't.

6.22 __int64 tts(__int64 time);

Converts clock tics into seconds.

6.23 __int64 ttm(__int64 time);

Converts clock tics into milliseconds.

6.24 __int64 ttmu(__int64 time);

Converts clock tics into microseconds.

6.25 __int64 stt(__int64 time);

Converts seconds into clock tics.

6.26 __int64 mtt(__int64 time);

Converts milliseconds into clock tics.

6.27 __int64 mutt(__int64 time);

Converts microseconds into clock tics.

7 Randomization

Defined in random.h and random.c

This part contains a number of randomization functions based upon an paper by Marc Brysbaert in BRMIC, 1992. More specifically:

- randomizes drawing from uniform, normal or exponential decreasing distributions.
- producing random lists without replacement.
- producing pseudo-random lists with limitations to the succession of values (for instance used for task-switching).

When working with complex randomization, it's better to use two programs: one to produce the randomized list and one to run the trials, based on the values in the randomized list. These are two completely different jobs and putting these together can complicate the testing of your program.

Another option is to use the statistical language R to produce random lists because it offers more sophisticated randomizers.

7.1 int ts_rseed (int newseed[3]);

Provides the possibility of user initiated random seeds. To be used for simulations: identical seeds always result in the same randomized list.

7.2 int ts_rseedfile (char *file);

Takes the seeds from a file. The last used seeds is always written to `seed.dat`. Starting a new random sequence with the seeds in this file can be used to reduce the chance of overlapping random series in sequential runs to zero. However, chances of overlapping series is by standard incredibly small so this function is to please the die-hard control freaks. There is even a risk in using this function: if you copy your program to different computers, without deleting `seed.dat`, all your computers will generate the same random lists.

7.3 double ts_runif ();

Returns a random double, drawn from a uniform distribution with minimum 0 and maximum 1. This is the basic random function used by all other functions below. The seeds are automatically initialized at the first call of the function. This initialization is based upon the clock unless the seeds are provided by the user by ts_rseed or ts_seedfile.

7.4 double ts_rexp ();

Returns a random double drawn from a distribution with exponential decreasing density. Minimum is 0 and mean is 1.

7.5 double ts_rnorm (double mean, double sd);

Returns a random double drawn from a normal distribution with mean mean and standard deviation sd (for a standard normal distribution use mean 0 and standard deviation 1).

7.6 int ts_rint (int nmax);

Returns a random integer from the specified range [0:nmax-1].

7.7 int ts_rlist (int nmax, int freq, int *list);

Fills *list with a list of random integers. This list contains all numbers from 0 to nmax-1 freq times.

Don't forget to provide an array large enough to contain the entire list (size = nmax x freq).

7.8 int ts_rslist (int nmax, int freq, int *list);

Fills *list with a list of random integers. This list contains all coupled combinations of the numbers 0 to nmax-1 freq times.

Don't forget to provide an array large enough to contain the entire list (size = (nmax x nmax x freq)+1).

8 Mouse support

Defined in mouse.h and mouse.c.

This is a small set of functions, that can be used for two purposes.

The first is giving instructions to participants. One function draws a box around some text (instructions) you generate with the standard text output functions, another function puts a click-to-continue-button on the screen.

The other mouse functions activate or hide a mouse pointer, and report the coordinates of the mouse pointer on the screen. Mouse button presses can be registered trough the standard response functions. These functions can be used to generate more complex user interfaces, or for example to implement a Corsi block tapping task (see examples).

8.1 void ts_textbox (int x1, int y1, int x2, int y2);

Draws a frame between points (x1,y1) and (x2,y2). Within this frame text can be put using the standard text output functions.

8.2 void ts_button (int x, int y);

Draws a button around point (x,y). Loads the mouse driver if necessary and draws a mouse-pointer to the screen. Left-clicking the button continues the program.

8.3 void ts_drawmouse();

Loads the mouse driver if necessary and draws a mouse pointer on the screen.

8.4 void ts_hidemouse();

Hides the mouse pointer.

8.5 int ts_xmouse();

Returns the horizontal position of the mouse pointer

8.6 int ts_ymouse();

Returns the vertical position of the mouse pointer

8.7 void ts_mousepos (int x, int y);

Moves the mouse pointer to position (x,y).

8.8 int ts_mousecolor (int color);

Changes the color of the mouse pointer. Returns the previous color value.

9 Bitmaps

Defined in blit.h and blit.c

Normally, all graphics functions draw onto the screen. It is also possible to draw onto a memory bitmap, and blit that bitmap to the screen, once your drawing operations are finished. This is necessary in cases where your stimulus is too complex to be drawn within one refresh cycle of the screen. The double buffer also provides similar functionality.

The bitmap functions add the following:

- it is possible to have more than one bitmap in memory (increase program speed even more)
- you can choose the size of the bitmap (save memory)
- you can load bitmaps from a .bmp file
- you can do special operations, like rotating, zooming, etc.

The destination coordinates of the blitting functions below are interpreted dependent on the coordinate system used. With the Cartesian coordinate system, bitmaps are blitted centered around the destination coordinates. With the standard coordinate system, the destination coordinates correspond to the upper left corner of the blitting area.

9.1 struct map;

Bitmap memory structure used by Tscope. It is defined as follows:

```
typedef struct {
    int sw, sh;
    int w, h;
    BITMAP *b;
} map;
```

sw and sh contain the total width and height of the bitmap in pixels.

w and h contain the with and height of one quadrant of the bitmap (for use with the Cartesian coordinate system).

b points to the content of the bitmap.

9.2 map *ts_makebmp (int w, int h);

Creates a bitmap with dimensions w and h.

If you use the Cartesian coordinate system, w and h are interpreted as the horizontal and vertical size of each quadrant of the bitmap. The center of the bitmap will be (0,0), x will range from -w to +w, y from -h to +h.

If you use the standard coordinate system, w and h are interpreted as the horizontal and vertical size of the whole bitmap. (0,0) will be the upper left corner of the bitmap, (w,h) will be the lower right of the bitmap.

The default background color of the newly created bitmap is the same as the screen and double buffer background color.

Returns the address of the newly created bitmap.

9.3 map *ts_readbmp (char *file);

Creates a bitmap containing the bitmap-file.

Returns the address of the bitmap.

9.4 void ts_killbmp (map *what);

Removes the specified bitmap out of memory. You should do this at the end of your program for all bitmaps created with ts_makebmp or ts_readbmp.

9.5 map *ts_tobmp (map *where);

Defines one the specified bitmap as destination of the following drawing operations.

9.6 void ts_clrbmp (map *what, int color);

Clears one of the self-made bitmaps. Color defines the background color. You can either provide a color (pre-defined or by means of ts_makecolor), or you can use the pre-defined TRANS and NONE values. TRANS makes the background transparent, while NONE takes the background color of the screen and double buffer.

9.7 void ts_blit (map *what, int dest_x, int dest_y);

Blits the specified bitmap to the current destination bitmap, at the position defined by (dest_x, dest_y).

9.8 void ts_partblit (map *what, int src_x1, int src_y1, int src_x2, int src_y2, int dest_x, int dest_y);

Blits part of the specified bitmap to the current destination bitmap. (src_x1, src_y1) en (src_x2, src_y2) are the coordinates of the rectangular part of the source bitmap. The image is blitted at position (dest_x, dest_y).

9.9 void ts_stretchblit (map *what, float ratio, int dest_x, int dest_y);

Blits a magnified or scaled down version of the specified bitmap to the destination bitmap at position (dest_x, dest_y). Ratio defines the scale factor (1 = original size, ¿1 magnify, ¡1 scale down).

9.10 void ts_flipblit (map *what, int flip, int dest_x, int dest_y);

Flips and blits the specified bitmap to the destination bitmap at position (dest_x, dest_y). With flip you provide type of flip. VFLIP flips the bitmap vertically, while HFLIP results in a horizontal flip. VHFLIP combines both.

9.11 void ts_rotateblit (map *what, int cx, int cy, float angle, int dest_x, int dest_y);

Rotates and blits the specified bitmap to the destination bitmap at position (dest_x, dest_y). (cx, cy) Defines the center of the rotation, angle defines the angle (0-360, counterclockwise).

10 Sound support

Defined in sound.h and sound.c

There are three groups of functions that provide sound support. The 'sample' functions are for playing sounds from a .wav file, the 'stream' functions generate simple sounds on the fly. The third group are the parameter functions for both.

10.1 SAMPLE *ts_loadsample (char *filename);

Loads a .wav file into memory.

Returns the address of the sample.

10.2 void ts_killsample (SAMPLE *spl);

Removes the specified sample out of memory. You should do this at the end of your program for all samples loaded with ts_loadsample.

10.3 int ts_playsample (SAMPLE *spl);

Starts playing the specified sample in the background (i.e. the function gives control back to your program right away, it does not wait until the end of the sample). Volume, pan, and whether the sample is looped or played only once are controlled by the parameter functions below.

A program running with REALTIME priority will not play samples.

Returns the voice number that was allocated for the sample or negative if no voices were available.

10.4 void ts_adjustsample (SAMPLE *spl);

Alters the parameters of a sample while it is playing (which is useful for manipulating looped sounds). Volume, pan and the loop flag can be manipulated. Normally, changing parameter values does not influence a playing sample, so the parameters have to be set before playing the sample. With this function you can impose the parameter changes on a sample while it is playing.

10.5 void ts_stopsample (SAMPLE *spl);

Stops playing the specified sample.

10.6 void ts_playstream(__int64 time);

Generates & plays a sound in the foreground (i.e. the function does not give control back to your program before the sound has stopped playing). After the specified time the sound stops and control goes back to your program.

The volume and pan of the sound are controlled by the parameter functions below.

The standard sound is a sine wave of 800Hz. The frequency of the wave can be changed with ts_sinefreq. You can replace the sine wave with a sound generating function of your own.

In contrast to the sample functions, the sound functions also work in programs running with REALTIME priority.

10.7 int ts_rtstream(__int64 * time, __int64 * error, __int64 maxtime);

Same as ts_playstream, but stops playing the sound if a response is given. Parameters and return value are the same as other response registration functions.

10.8 int ts_volume (int volume);

Sets the volume of samples or sounds to be played. Accepts values between 0 and 255. Default is 255.

The values are on a linear scale, our ears' sensitivity is logarithmic. This means the audible difference between values 1 and 2 will be about the same as the difference between 100 and 200.

The output volume also depends on your Windows's volume settings. If these are set to 0, you will hear nothing, even if you set the ts_volume to its maximum. `sndvol32.exe` controls the Windows sound volume (choose start menu - run, and type sndvol32 to open it). Both 'Volume control' and 'Wave' should be set to the maximum for optimal results. The balance should be set to the center.

Returns the previous value.

10.9 int ts_pan (int pan);

Sets the pan of samples or sounds to be played. Accepts values between 0 (all sound to the left) and 255 (all sound to the right). Default is 127 (centered).

The panning also depends on your Windows's balance settings. See the previous entry for more information about setting the balance in Windows.

Returns the previous value.

10.10 int ts_loop (int loop);

Controls whether samples will be looped or played only once. Accepts ON and OFF as argument. Returns the previous value.

10.11 int ts_sinefreq (int freq);

Sets the frequency of the sine wave function. Default is 800. Maximum is half of the samplerate (see below).

Returns the previous value.

10.12 unsigned char (*ts_streamfunc(unsigned char (*func) (__int64))) (__int64);

Replace the sine function with your own sound generating function.

10.13 void ts_drawsound ();

Helper function that draws a graphical representation of the values generated by the sound function currently in use.

10.14 int ts_streambufsize (int size);

Sets the size of the stream buffer used by the sound streams. Default is 1024 bytes. Increase if there are cracks in the sound. Decrease if the sound doesn't stop playing fast enough after a response or after the end of the playing interval.

Returns the previous value.

10.15 int ts_samplerate (int rate);

Sets the number of samples per second. The samplerate influences the sound quality (higher is better). The maximum sound frequency that can be achieved is half of the samplerate. A rate of 44100 samples per second is cd quality, 11025 is telephone quality. Default is 22050.

Returns the previous value.

Returns a pointer to the previous sound function.

10.16 AUDIOSTREAM * ts_makestream();

Makes an audio stream using the current samplerate, streambuffer size, volume and pan.

Returns the address of the newly created audio stream.

10.17 void ts_killstream (AUDIOSTREAM *stream);

Removes the specified audio stream from memory. You should do this at the end of your program for all streams created with ts_makestream.

10.18 void ts_updatestream (AUDIOSTREAM *stream);

Fills the buffer of the specified audio stream with values generated by the current sound generating function.

11 Sound support - new (experimental) API

Defined in sound2.h and sound2.c

The old sound functions work fine for playing sounds, but recording sound does not work with the underlying allegro library. Therefore a new set of sound functions are being implemented based on the libsndfile and portaudio libraries.

The functions are still experimental. This has some implications:

- currently the new sound functions start with snd2_ instead of the usual ts_. This will change once all features are implemented and all (or most) bugs are found.
- the set of functions is not complete yet. Expect additions.
- the behavior and interface of the functions may change.
- there may be bugs.

The old sound functions still work, but you can not mix systems in one program. Tscope will exit (at runtime) when you try to do that.

11.1 struct snd2_sample;

Bitmap memory structure used by Tscope. It is defined as follows:

```
typedef struct {
int channels;
int samplerate;
sf_count_t frames;
sf_count_t current_frame;
float *data;
} snd2_sample;
```

channels and samplerate contain the number of channels and the sample rate of the sample.

frames and current_frame contain the total number of frames of the sample and the frame that is currently being processed (for internal use).

data points to the contents of the sample.

11.2 struct snd2_stream;

A sound stream.

11.3 int snd2_channels(int channels);

Sets the number of channels the recording and stream functions. Possible values are 1 (MONO) or 2 (STEREO). Default is 1.

Has no influence on samples that are read from disk - for disk files the number of channels is set by the file's header.

Returns the previous number of channels.

11.4 int snd2_samplerate(int rate);

Sets the number of samples per second for the recording and stream functions. The samplerate influences the sound quality (higher is better). Supported values are 44100 (cd quality), 22050 or 11025. Default is 22050. This is good enough for recording voices.

Has no influence on samples that are read from disk - for disk files the sample rate is set by the file's header.

Returns the previous samplerate.

11.5 int snd2_sampleformat(int format);

Sets the sample format in which samples are written to disk. Possible values are 16-bit integer (SAMPLE_INTEGER) and 32-bit floating point (SAMPLE_FLOAT). Default is SAMPLE_INTEGER.

Internally (i.e. in RAM memory) samples are always represented as 32-bit floats.

Returns the previous sample format.

11.6 snd2_sample *snd2_makesample(int length);

Creates a sample that is large enough to hold 'length' milliseconds of sample data and sets the samplerate and number of channels according to the values set by snd2_samplerate and snd2_channels. The sample data is initialized to 0 and the current frame set to 0.Returns a pointer to the sample that was created.

11.7 void snd2_allocatesample(snd2_sample *samp);

Internal function. Allocates memory for the data element of the snd2_sample stucture and initializes the data to 0. The amout of memory is set according to the frames element of the snd2_sample structure.

11.8 void snd2_killsample (snd2_sample *samp);

Removes the specified sample out of memory. You should do this at the end of your program for all samples generated with snd2_makesample or snd2_readsample.

11.9 snd2_sample *snd2_readsample (char *file);

Reads a sample from the specified file and copies it to the sample structure.Returns a pointer to the sample that was created.

11.10 int snd2_writesample(snd2_sample *samp, char *file);

Writes the sample to the specified file.Returns the number of frames written.

11.11 int snd2_recordsample_blocking(snd2_sample *samp);

Records the sample samp using the blocking api. The function does not return until the samples data element is full, i.e. until the number of frames specified in the frames element of the sample structure have been recorded.

11.12 void snd2_playsample_blocking(snd2_sample *samp);

Plays the sample samp using the blocking api. The function does not return until the sample is finished, i.e. until the number of frames specified in the frames element of the sample structure have been played.

11.13 snd2_stream * snd2_recordsample(snd2_sample *samp);

Starts recording the sample samp using the non-blocking api. The function returns immediately and the sample is recorded until its data element is full or until snd2_stopsample is called.Returns a pointer to the stream that was started.

11.14 snd2_stream *snd2_playsample(snd2_sample *samp);

Starts playing the sample samp using the non-blocking api. The function returns immediately and the sample is played until it is finished or until snd2_stopsample is called.Returns a pointer to the stream that was started.

11.15 int snd2_querysample(snd2_stream * stream)

Checks whether a given stream has finished playing/recording or not. Returns 1 when the stream is still active and 0 when the stream has finished.

11.16 void snd2_stopsample(snd2_stream * stream);

Stops playing/recording a given stream.

11.17 float snd2_getstreamtime(snd2_stream * stream);

returns the time of a given stream in seconds.

11.18 __int64 snd2_getsampletime(snd2_sample * samp);

returns the time of a given sample in samples.

11.19 void snd2_tobuffer(snd2_sample * samp);

dumps the contents of a sample to disk.

11.20 void snd2_frombuffer(snd2_sample *samp);

reads the contents of a the disk buffer to a sample.

12 Linux notes

All differences between the Windows and Linux version of Tscope are reported here. If a function is not mentioned in this section, it will behave identical on both operating systems.

12.1 ts_priority

From tscope version 168 onwards this function can actually increase the priority of your program when running on Linux. It must, however, be used with care. Here are some considerations:

- On Linux systems the higher priority modes can only be selected when running as root. When a non-root user requests a higher priority the program writes a warning to the console and continues running with normal priority.
- When running in X11 mode, increasing the priority of your program will slow down the X11 server. Therefore, it is necessary to suspend your program from time to time to allow the X11 server to display your stimuli, poll the mouse status, etc. The function ts_waitmode controls where and when Tscope will suspend your program. You will need to experiment with the different wait modes to find a good setting. Giving up too much cpu cycles will cause the vsync function to miss sync signals, keeping all cycles for yourself will slow down graphics.
- Running in console mode (as root) seems the best option when running programs with increased priority. The only problem is that the current version of Allegro has trouble with usb mice on linux. On my system this could be fixed by telling Allegro to read the mouse state from /dev/input/eventN, by adding

  ```
  [mouse]
  mouse = EV
  ```

  to the allegro configuration file. This configuration file can be found in the source directory of allegro and is called allegro.cfg. If you are using ts_scrcfg and ts_timercfg you will need to add this to your tscope.cfg file.
- The good news, however, is that timing precision on Linux is already quite accurate when running with normal priority in X11 mode. Moreover, as in the Windows version, the *error parameter of the timing functions will accurately report the timing error of each timed event, so that trials where the timing error was too large can be filtered out afterwards.

Currently, Tscope uses a call to the function *setpriority* to change the priority of the program. The priority of a program could be uncreased even further by calling *sched_setscheduler* and *mlockall*, but this would also increase the risk of locking up your system.

12.2 ts_waitmode

On Linux, Tscope programs can give up short intervals of cpu time by blocking on a read of the real time clock (/dev/rtc). This allows other processes (such as X11) to run for a short time. When running a program with higher priority in X11 mode, the waitmode should be set to SLEEP_WAIT. This can, however, only be done when running as root. If a waitmode different from the default BUSYLOOP is requested by a non-root user, the program will write a warning to the console and will continue using the BUSYLOOP waitmode.

12.3 ts_defkey

Most input devices will work in the same way on Linux and on Windows. There are two minor differences:

- The game and parallel port can only be read when running as root.
- The cedrus usb serial response box is not supported on Linux.

12.4 ts_vsync

The logic for selecting a vsync mode differs from the Windows version for the following reasons:

- Only root can read the status of the vsync signal from the registry.
- The actual refresh rate can not be reported by Linux. In order to know the refreshrate of your screen, it is necessary to start your program with the desired graphics mode and then use your screen's menu to read out the refresh rate. This value can then be entered in your call to ts_refreshrate to make the vsync simulation work properly.

Under ideal circumstances, the function reads the status of vertical retrace signal directly from the computer's registry. In situations where that's not possible, a vertical retrace signal is simulated.

Tscope will simulate the retrace signal at the requested refresh rate in the following cases:

- If no window is opened yet.
- In windowed mode.
- If the refresh rate is too high. Default the maximum value is 70Hz, but it can be altered by the function ts_vsynclimit.
- If the vsync register cannot be opened. (i.e. if the user is not root).

13 Mac OS X notes

All differences between the Windows and Mac OS X version of Tscope are reported here. If a function is not mentioned in this section, it will behave identical on both operating systems.

13.1 ts_priority

On Mac, the priority of a Tscope program can not be changed. All programs will run with normal priority on Mac. If the user tries to change the priority a warning will be printed on the console and the program will continue.

13.2 ts_waitmode

On Mac, the waiting mode of a Tscope program can not be changed. All programs will wait using a busyloop on Mac. If the user tries to change the waiting mode a warning will be printed on the console and the program will continue.

13.3 ts_defkey

Only the keyboard, mouse and joystick work on Mac. Reading input from the parallel, game or serial port is not possible. If a user tries to define one of these keys the program will abort with an error message.

13.4 ts_vsync

On Mac, the status of the vsync signal can not be read directly from the register. Tscope will always revert to simulating a vsync signal at the requested refreshrate on Mac. If the user tries to change the vsync mode a warning will be printed on the console and the program will continue.

13.5 ts_trigger

On Mac, Tscope can not write directly to the parallel port. If the user tries to send trigger signals to the parallel port Tscope will close with an error message.

14 Appendix A: Predefined values

14.1 Debug levels

- DEBUG0
- DEBUG1
- DEBUG2
- DEBUG3
- DEBUG4

14.2 Color depths

- 8, 15, 16, 24 or 32.

14.3 Screen sizes

Normal aspect ratio:

- SIZE0 (QVGA, 320 x 240 pixels).
- SIZE1 (VGA, 640 x 480 pixels).
- SIZE2 (SVGA, 800 x 600 pixels).
- SIZE3 (XGA, 1024 x 768 pixels).
- SIZE4 (XGAplus, 1152 x 864 pixels).
- SIZE5 (SXGA, 1280 x 1024 pixels).
- SIZE6 (SXGAplus, 1400 x 1050 pixels).
- SIZE7 (UXGA, 1600 x 1200 pixels).

Widescreen aspect ratio:

- WSIZE3 (WXGA, 1280 x 768 pixels).
- WSIZE4 (WXGAplus, 1440 x 900 pixels).
- WSIZE5 (WSXGA, 1600 x 1024 pixels).
- WSIZE6 (WSXGAplus, 1680 x 1050 pixels).
- WSIZE7 (WUXGA, 1920 x 1200 pixels).

14.4 Screen modes

- WINDOW
- FULLSCREEN
- FULLSCREEN_ACCEL

14.5 Colors

- WHITE
- BLACK
- RED
- GREEN
- BLUE
- YELLOW
- MAGENTA
- CYAN
- GREY75
- GREY50
- GREY25

14.6 Fonts

- ARIAL
- COURIER
- TIMES

14.7 Font sizes

- 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72.

14.8 Font styles

- REG
- BOLD
- IT
- BOLDIT

14.9 Coordinate systems

- CARTESIAN
- STANDARD

14.10 Program priorities

- REALTIME
- HIGH
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- IDLE

14.11 Buttons

- **Keyboard**
  - KEY_A, KEY_B, KEY_C, KEY_D, KEY_E, KEY_F, KEY_G, KEY_H, KEY_I, KEY_J, KEY_K, KEY_L, KEY_M, KEY_N, KEY_O, KEY_P, KEY_Q, KEY_R, KEY_S, KEY_T, KEY_U, KEY_V, KEY_W, KEY_X, KEY_Y, KEY_Z.
  - KEY_0, KEY_1, KEY_2, KEY_3, KEY_4, KEY_5, KEY_6, KEY_7, KEY_8, KEY_9.
  - KEY_0_PAD, KEY_1_PAD, KEY_2_PAD, KEY_3_PAD, KEY_4_PAD, KEY_5_PAD, KEY_6_PAD, KEY_7_PAD, KEY_8_PAD, KEY_9_PAD.
  - KEY_F1, KEY_F2, KEY_F3, KEY_F4, KEY_F5, KEY_F6, KEY_F7, KEY_F8, KEY_F9, KEY_F10, KEY_F11, KEY_F12.
  - KEY_ESC, KEY_TILDE, KEY_MINUS, KEY_EQUALS, KEY_BACKSPACE, KEY_TAB.
  - KEY_OPENBRACE, KEY_CLOSEBRACE, KEY_ENTER, KEY_COLON, KEY_QUOTE.
  - KEY_BACKSLASH, KEY_BACKSLASH2, KEY_COMMA, KEY_STOP, KEY_SLASH.

  - KEY_SPACE, KEY_INSERT, KEY_DEL, KEY_HOME, KEY_END, KEY_PGUP, KEY_PGDN.
  - KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN.
  - KEY_SLASH_PAD, KEY_ASTERISK, KEY_MINUS_PAD, KEY_PLUS_PAD, KEY_DEL_PAD, KEY_ENTER_PAD.
  - KEY_PRTSCR, KEY_PAUSE, KEY_ABNT_C1.
  - KEY_YEN, KEY_KANA, KEY_CONVERT, KEY_NOCONVERT, KEY_AT, KEY_CIRCUMFLEX, KEY_COLON2, KEY_KANJI.
  - KEY_LSHIFT, KEY_RSHIFT, KEY_LCONTROL, KEY_RCONTROL, KEY_ALT, KEY_ALTGR, KEY_LWIN, KEY_RWIN, KEY_MENU.
  - KEY_SCRLOCK, KEY_NUMLOCK, KEY_CAPSLOCK.
- **Keyboard, from cygwin4tscope-1.0.3 / allegro 4.2.0 onwards**
  - KEY_EQUALS_PAD, KEY_BACKQUOTE, KEY_SEMICOLON, KEY_COMMAND.
  - KEY_UNKNOWN1, KEY_UNKNOWN2, KEY_UNKNOWN3, KEY_UNKNOWN4.
  - KEY_UNKNOWN5, KEY_UNKNOWN6, KEY_UNKNOWN7, KEY_UNKNOWN8.
- **Parallel port 1:** P1, P2, P3, P4, P5.
- **Parallel port 1 (inverted buttons):** IP1, IP2, IP3, IP4, IP5.
- **Parallel port 2:** PP1, PP2, PP3, PP4, PP5.
- **Parallel port 2 (inverted buttons):** IPP1, IPP2, IPP3, IPP4, IPP5.
- **Parallel port 3:** PPP1, PPP2, PPP3, PPP4, PPP5.
- **Parallel port 3 (inverted buttons):** IPPP1, IPPP2, IPPP3, IPPP4, IPPP5.
- **Game port:** G1, G2, G3, G4.
- **Game port (inverted buttons):** IG1, IG2, IG3, IG4.
- **Cedrus serial port emulation on usb:** S1, S2, S3, S4, S5, S6, S7.
- **Joystick:** J1, J2, J3, J4, J5, J6, J7, J8, J9, J10.
- **Mouse:** M1, M2, M3.

14.12 Wait modes

- BUSYLOOP
- SLEEP_WAIT
- SLEEP_RESP
- SLEEP_VSYNC

14.13 Vsync modes

- REALSYNC
- SIMSYNC_ACTUAL
- SIMSYNC_REQ
- WHATEVER

14.14 Flipped blit modes

- VFLIP
- HFLIP
- VHFLIP

14.15 Sound: number of channels

- MONO
- STEREO

14.16 Sound: sample format for output files

- SAMPLE_INTEGER
- SAMPLE_FLOAT

15 Appendix B: Tscope internals

Defined in internal.h and internal.c.

These functions initialize and close the devices, subsystems etc. used by Tscope. Normally, each device or subsystem is initialized automatically at the moment it is needed, so you won't have to bother calling these functions. In fact, you just can't - they are not declared in tscope.h.

One of the moments where you might need to use these functions, is when you are writing an add-on package for Tscope, rather than a program that just uses Tscope functions. The variables and functions defined here can be accessed if you include tscope/internal.h in your program.

The functions and variables from internal.c and .h are documented in the header file, not over here.